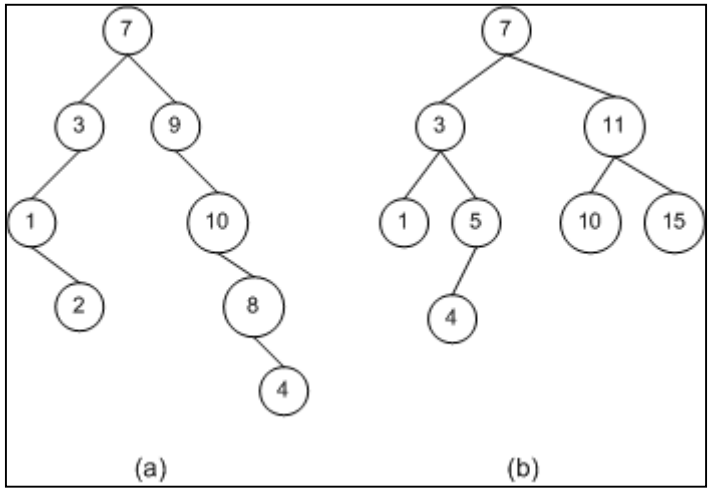
Lecture 8

Binary search tree (BST) operations

Binary search tree

- A *binary search tree* is a special kind of binary tree designed to improve the efficiency of searching through the contents of a binary tree.
- Binary search trees exhibit the following property:
    - for any node *n*, every descendant node's value in the left *sub-tree* of *n* is less than the value of *n*, and every descendant node's value in the right *sub-tree* is greater than the value of *n*.
- A sub-tree rooted at node *n* is the tree formed by imaging node *n* was a root. That is, the sub-tree's nodes are the descendants of *n* and the sub-tree's root is *n* itself.

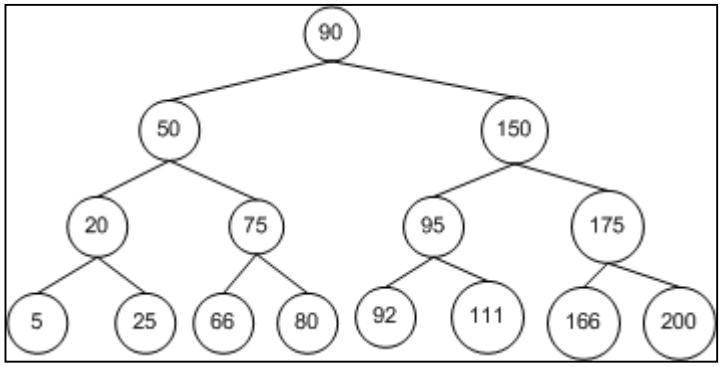


a is not a BST, b is a BST

searching a binary search tree

BST searching is as follows. We have a node *n* we wish to find (or determine if it exists), and we have a reference to the BST's root. This algorithm performs a number of comparisons until a null reference is hit or until the node we are searching for is found. At each step we are dealing with two nodes: a node in the tree, call it *c*, that we are currently comparing with *n*, the node we are looking for. Initially, *c* is the root of the BST. We apply the following steps:

1. If *c* is a null reference, then exit the algorithm. *n* is not in the BST.
2. Compare *c*'s value and *n*'s value.
3. If the values are equal, then we found *n*.
4. If *n*'s value is less than *c*'s then *n*, if it exists, must be in the *c*'s left sub-tree. Therefore, return to step 1, letting *c* be *c*'s left child.
5. If *n*'s value is greater than *c*'s then *n*, if it exists, must be in the *c*'s right sub-tree. Therefore, return to step 1, letting *c* be *c*'s right child.



- Practice searching for 5,25,66,95
- Is the search time is linear?
- What relation between the search time and the number of elements in the tree
- Is the distribution of the elements in the binary tree get some importance when searching it

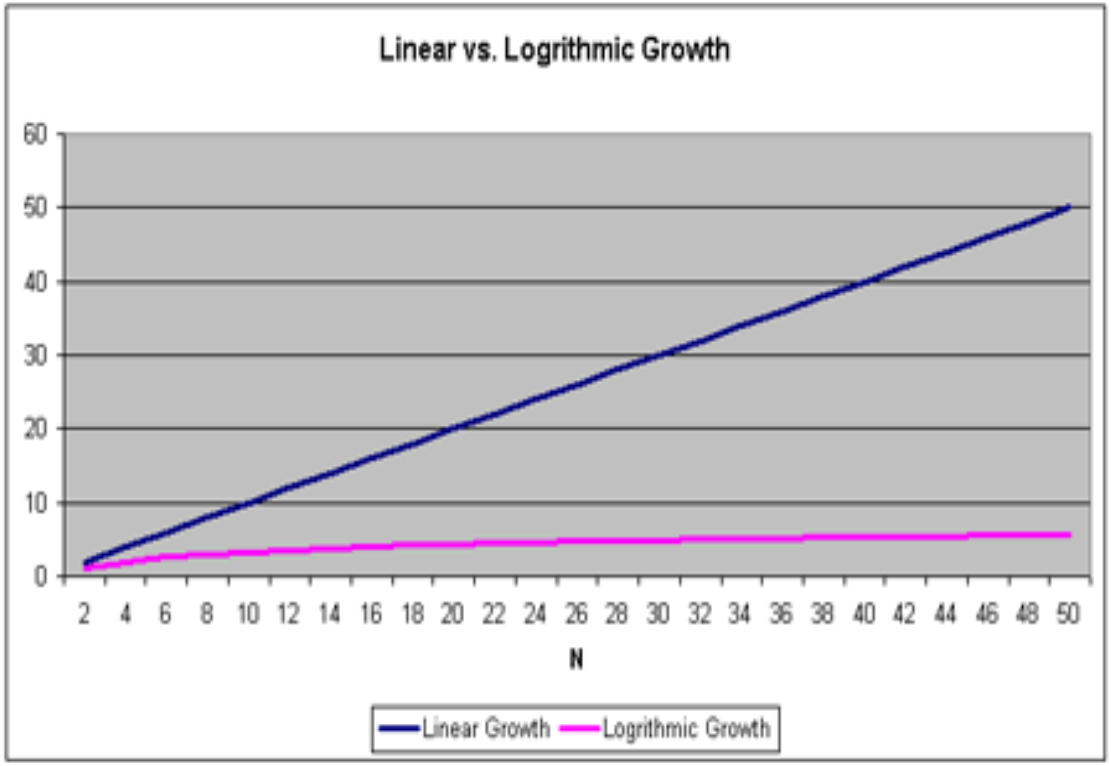
BST searching time

After each comparison, the search domain is reduced by one half.
Assume the worst case (the element we search for is one of the deepest element in a balanced BST of N elements
One comparison reduces the domain to N/2
The second reduces it to N/4
The third reduces it to N/8
This continues until the domain has only one element that requires the last comparison

How many comparison, for N=8,16,32?
Could you deduce how many comparisons for N?
BigOh notation: Search a Balanced BST is $O(\log_2 N)$

- Search an element in a non-sorted array is O(……..)
    - Write the code
    - Count the main operation

Linear versus Logarithmic search time



BST search implementation

```
public class BinarySearchTree<T>
{
    private BinaryTreeNode<T> root;
    int count = 0;
    private IComparer<T> comparer;
    public BinarySearchTree(IComparer<T> comparer)
    {
        root = null;
        this.comparer = comparer;
    }
    public virtual void Clear()
    {root = null;}
    public BinaryTreeNode<T> Root
    {
        get{return root;}
        set{root = value;}
    }
    public bool Contains(T data)
    {
        // search the tree for a node that contains data
        BinaryTreeNode<T> current = root;
        int result;
        while (current != null)
        {
            // defined as a private member in the tree class Comparer<T>
            result = comparer.Compare(current.Value, data);
            if (result == 0) return true;// we found data
            // current.Value > data, search current's left subtree
            else if (result > 0) current = current.Left;
            // current.Value < data, search current's right subtree
            else if (result < 0) current = current.Right;
        }
        return false;       // didn't find data
    }
    public virtual void Add(T data)...
    public bool Remove(T data,  Comparer<T> comparer)...
}
```

Report Discussion

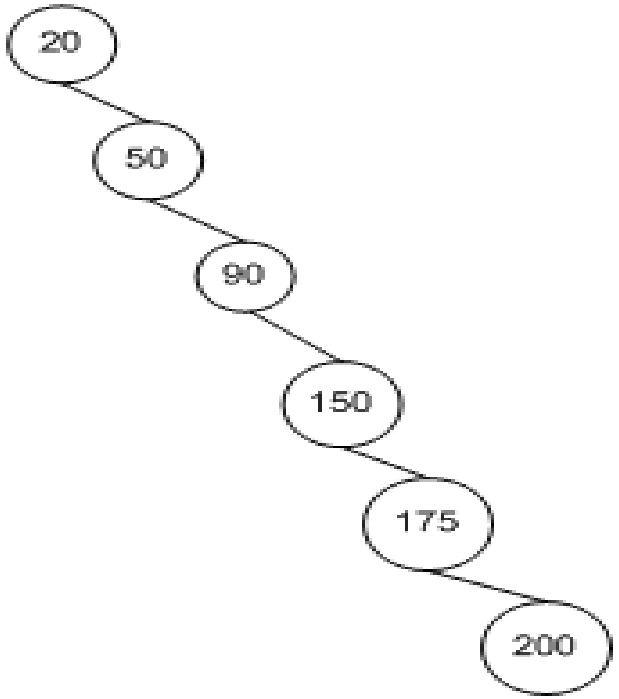
Last lecture report:

- Does the order in which the nodes added to the binary tree affect the search time for a node with a given value?
- Give some demonstrating examples

New report: Deduce the BigO for searching a node in a Binary search tree

Topology of the binary tree and the search time

- Is the shown tree is a BST?
- What is the BigOh search time when search the element at the last node?
- What is your conclusion?

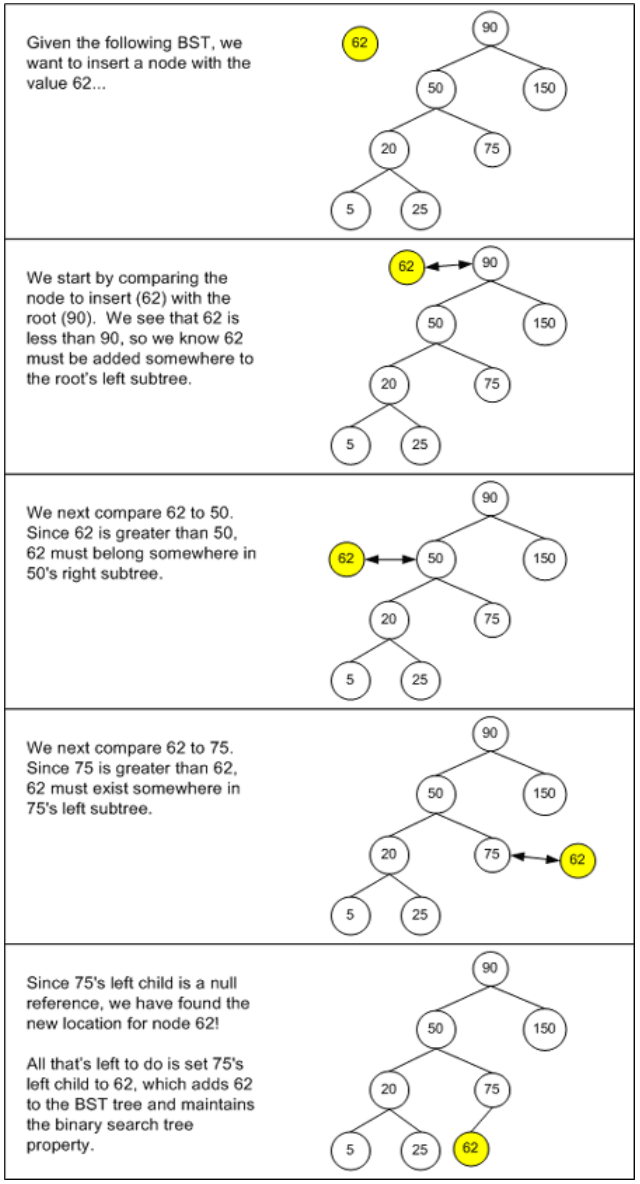


The searching time in a binary search tree depends on the topology of the tree. In an ideal case (balanced BST where the mid-point is the root) the searching is $O(\log_2 n)$. In the worst case (the one shown here) it's a linear search

Inserting a node in the BST

The insertion algorithm is as follows, we'll be making comparisons between a node *c* and the node to be inserted, *n*. We'll also need to keep track of *c*'s parent node. Initially, *c* is the BST root and *parent* is a null reference. Locating the new parent node is accomplished by using the following algorithm:

1. If *c* is a null reference, then *parent* will be the parent of *n*. If *n*'s value is less than *parent*'s value, then *n* will be *parent*'s new left child; otherwise *n* will be *parent*'s new right child.
2. Compare *c* and *n*'s values.
3. If *c*'s value equals *n*'s value, then the user is attempting to insert a duplicate node. Either simply discard the new node, or raise an exception. (Note that the nodes' values in a BST must be unique.)
4. If *n*'s value is less than *c*'s value, then *n* must end up in *c*'s left subtree. Let *parent* equal *c* and *c* equal *c*'s left child, and return to step 1.
5. If *n*'s value is greater than *c*'s value, then *n* must end up in *c*'s right subtree. Let *parent* equal *c* and *c* equal *c*'s right child, and return to step 1.



Implementing node addition in a BST

What is the BigOh of the addition?

```
public virtual void Add(T data)
{
    // create a new Node instance
    BinaryTreeNode<T> n = new BinaryTreeNode<T>(data);
    int result;
    // now, insert n into the tree
    // trace down the tree until we hit a NULL
    BinaryTreeNode<T> current = root, parent = null;
    while (current != null)
    {
        result = comparer.Compare(current.Value, data);
        // they are equal - attempting to enter a duplicate - do nothing
        if (result == 0) return;
        // current.Value > data, must add n to current's left subtree
        else if (result > 0){parent = current; current = current.Left;}
        // current.Value < data, must add n to current's right subtree
        else if (result < 0){parent = current;current = current.Right;}
    }
    // We're ready to add the node!
    count++;// defined as a private field in the Tree class
    if (parent == null)
        root = n;// the tree was empty, make n the root
    else
    {
        result = comparer.Compare(parent.Value, data);
        // parent.Value > data, therefore n must be added to the left subtree
        if (result > 0) parent.Left = n;
        // parent.Value < data, therefore n must be added to the right subtree
        else parent.Right = n;
    }
}
```

What affect the topology of the BST

You knew that search a balanced BST is $O(\log_2 n)$ and it increases to linear search time in unbalanced BST

When inserting nodes in a BST we are controlled by the tree properties
The insertion algorithm itself has no control to make the tree balanced or not

Try making a BST from 1, 2, 3, 4, 5, and 6
Then try making a BST from the same set of numbers by ordered as 4, 2, 5, 1, 3, 6
Compare the topology of the resulting two BSTs

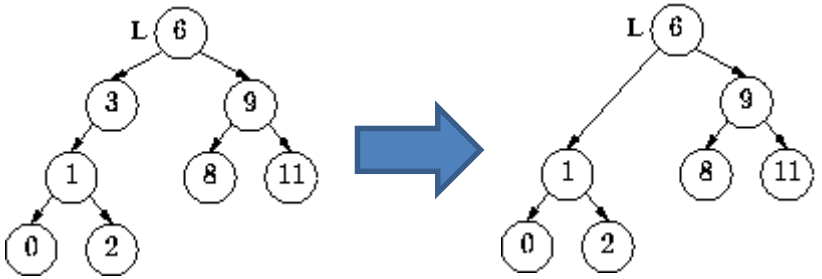
What makes it balanced?

The order in which the nodes are inserted in the BST determines if we get a balanced tree or not

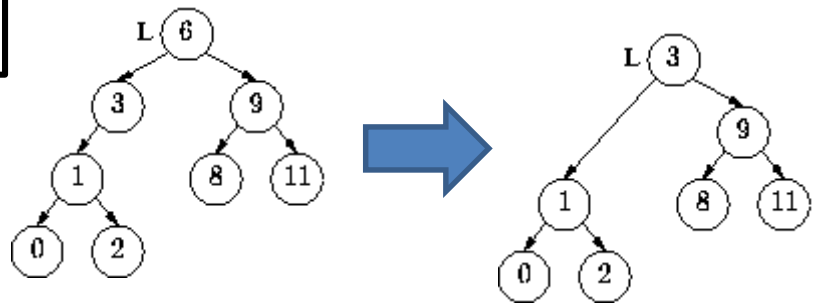
Deleting a node from the BST

Case 1: If we are trying to delete a leaf, there is no problem. We just delete it and the rest of the tree is exactly as it was, so it is still a BST

Case 2: The node we're deleting has only one sub tree. In the following example, `3' has only 1 sub tree. We just `link past' the node, i.e. connect the parent of the node directly to the node's only sub tree. This always works, whether the one sub tree is on the left or on the right.



Case 3: Deleting a node with two sub-trees (node 6)
Replace the value of the node to be deleted by the value of the greatest node in the left sub-tree
Delete the node with the greatest value in the left sub-tree
Or
The same with the right sub-tree but in choosing the smallest value



Deleting a node from the BST: Another example

The algorithm is self reading for the challengers